Abstraction without regret in database systems building: a manifesto

Christoph Koch, EPFL

Thoughts on joint work with:

Yanif Ahmad[+], Hassan Chafi[**], Thierry Coppey[*], Mohammad Dashti[*], Vojin Jovanovic[*], Oliver Kennedy[#], Yannis Klonatos[*], Milos Nikolic[*], Andres Noetzli[*], Martin Odersky[*], Tiark Rompf[*,**], Amir Shaikhha[*]

[*]EPFL [**]Oracle Labs [+]Johns Hopkins University [#]SUNY Buffalo

Abstract

It has been said that all problems in computer science can be solved by adding another level of indirection, except for performance problems, which are solved by removing levels of indirection. Compilers are our tools for removing levels of indirection automatically. However, we do not trust them when it comes to systems building. Most performance-critical systems are built in low-level programming languages such as C. Some of the downsides of this compared to using modern high-level programming languages are very well known: bugs, poor programmer productivity, a talent bottleneck, and cruelty to programming language researchers. In the future we might even add suboptimal performance to this list. In this article, I argue that compilers can be competitive with and outperform human experts at low-level database systems programming. Performance-critical database systems are a limited-enough domain for us to encode systems programming skills as compiler optimizations. However, mainstream compilers cannot do this: We need to work on optimizing compilers specialized for the systems programming domain. Recent progress makes their creation eminently feasible.

HAVE THE CAKE AND EAT IT TOO. *Abstraction without regret* refers to high level programming without performance penalty [RO12, Koc13]. The **main thesis of this article** is that databases implemented in high-level programming languages can beat state-of-the-art databases (implemented in C) performance wise.

MONUMENTS TO ABSTRACTION FAILURE. Let us look at the core of a classical SQL DBMS. On a high level of abstraction, textbook-style, we see nice, cleanly separated components, such as a storage manager, a buffer manager, a concurrency control subsystem, and a recovery manager. On a high level, the interfaces between these components are relatively clear, or are they?

In practice, the code of this core is a great monolith, formidable in its bulk (millions of lines of code). Buffer and storage management are tightly integrated. The page abstraction is a prime example of abstraction failure, and stores identifiers relevant to other subsystems, such as recovery. Concurrency control rears its ugly head when it comes to protecting ACID semantics against insertions and deletions, and when there are multiple access paths to data, some hierarchical like B-trees. In practice, concurrency control code is interspersed throughout the core. It is hard to even place concurrency control algorithms in their separate source code files; much is inlined in source files of the page abstraction, B-trees, and such. Physical logging breaks the abstractions of pretty much each of these subsystems. Sadly, this paragraph could be continued much longer.

WHY THE MONOLITH TODAY? The existence of the Monolith is a consequence of the use of C for building such systems. Modern programming languages offer features to create abstractions and protect them from leaking. We have powerful type systems and a strong arsenal of software composition primitives at our disposal, such as modules, interfaces, OO features (classes, objects, encapsulation, etc.), and genericity. Design patterns show us how to use these language features to solve abstraction problems and compose software in a

Preprint. To appear in IEEE Data Engineering Bulletin 37(1), March 2014.

maintainable way. Can modern high-level programming languages disentangle the Monolith and reclaim clean abstractions and interfaces? Of course: As Butler Lampson (and earlier David Wheeler) [Lam93] has claimed, *any problem in computer science can be solved by an additional level of indirection.* In the context of structuring software artifacts and eliminating abstraction leaks, this statement is true.

The problems come when performance matters. As Lampson's law suggests, these modern software composition concepts increase indirection, which compromises performance. It is the thesis of this article that compilers can eliminate these indirections automatically and reclaim efficiency; so we database people can dismantle the Monolith and do not need to keep living in PL stone age (pun intended).

BENEFITS OF HIGH-LEVEL PROGRAMMING. The main benefit of high-level programming languages is much increased productivity. The same functionality tends to be implementable in fewer lines of code[1], which improves maintainability and agility of software development. The improved productivity results in part from the fact that advanced language features such as genericity allow for more reusable code. Software libraries have better coverage of the needs of programmers than do libraries of low-level languages. Programmers approach software development with a different mindset: Resigned from the hope of the last bit of efficiency when using a high-level language, they reuse library code even when a re-implementation might yield slightly better performance. For example, in building systems in C with an emphasis on efficiency, we often feel the urge to build that additional list or tree data structure from scratch. In high-level programming, elegant generic collection classes readily available are just too attractive, and this rarely happens.

High-level programming languages, particularly functional ones, also tend to be better suited for automatic program analysis and transformation because analysis is localized and search spaces are smaller. One notable reason why this is true is that code that follows style conventions avoids hard-to-analyze counter arithmetics for core tasks such as looping through data structures, and uses abstractions such as map and foreach instead. Of course, it is much easier to robustly fuse or reorder foreach loops than C-style for loops with a loop counter. As a consequence, compilers can perform more powerful optimizations and transformations on high-level programs than on low-level ones. Fortunately, we can conclude from successful recent programming language designs such as Python and Scala that modern high-level programming languages are functional. Even Java is currently becoming a functional language [Jav13].

SUCCESS STORIES. Some recent successes of high-level programming in systems include the Singularity operating system from Microsoft Research [LH10] and Berkeley's Spark [ZCD+12]. Singularity is an operating system implemented almost in its entirety in a dialect of C#, and runs as managed code in Microsoft's CLR virtual machine. Still it outperforms other operating systems that are written in highly optimized C code. This is achieved not by a superior compiler but by program analysis and transformation techniques that would not be possible in C. For example, the source code of processes is analyzed for their potential for safe merging: such *software-isolated processes (SIPs)* are protected from each other using programming language constructs such as types only, but are not isolated by the operating system at runtime. A group of merged SIPs runs as a single process, and the performance benefit stems from the reduced amount of process switching, which is very expensive in classical operating systems.

Spark provides (fault-tolerant) distributed collections in Scala. It is easy to build map-reduce like systems, with improved performance compared to Hadoop and a much smaller code base (14000 lines of code [ZCD+12]). This is an example of how the improved productivity of high-level languages frees developer's cycles, which can be channeled into better designs. Spark benefits from Scala's functional nature by using closures, which are serialized and stored (for logical logging) as well as used to ship work between nodes.

These are examples of how high-level optimizations can dominate the cost of high-level programming. This should not come as a surprise; it is not foreign to database experience. Take the field of query processing. Optimizing join orders matters more than the choice of query operators. Algorithmic choices, such as choosing a good join operator, matter more than low-level implementation decisions. The choice of programming language generally has just a constant factor impact on performance. Performance is dominated by algorithmic and architectural decisions.

[1]The folklore has it that the improvement can be by up to two orders of magnitude, see also [Pyt14, Lea13].

COSTS OF HIGH-LEVEL PROGRAMMING. Complex software implemented in high-level programming languages makes extensive use of composition from smaller components. This includes components from previously existing libraries and components developed specifically as part of the new software artifact. Clean designs lead to deep nesting of class and interface hierarchies (both in terms of aggregation and inheritance/subtyping). Objects are copied rather than referenced to protect encapsulation. Abstraction requires complex and deep aggregations of containers, and thus the frequent creation, copying, and destruction of objects at runtime. Even basic types such as integers get boxed in objects for use in generic containers. Function call stacks are very deep, with correspondingly high overheads due to the creation of activation records and such[2]. Depending on how classes and generic code are instantiated, large amounts of code may be duplicated; much library code remains unused at runtime[3]. Functional programming languages in particular increase the amount of auxiliary object creation and destruction (boxing and unboxing) to impressive levels. Naïve compilers and runtime systems may see hundreds of object creations and destructions for a single constant-time call in a rich library[4]. Of course, modern (just-in-time) compilers such as Java Hotspot use advanced techniques to reduce boxing and unboxing, and runtime systems are optimized to handle the objects that cannot be avoided with relatively little overhead.

THE USE OF LOW-LEVEL LANGUAGES IN SYSTEMS BUILDING. Truth be told, compilers that eliminate all abstraction costs of developing a DBMS in a high-level programming language are not readily available today, but can be built, and with less effort than a classical DBMS. I will discuss how below.

The absence of suitable compilers is not a conclusive explanation for the use of C. If we started building a DBMS from scratch today, the rational approach would be to do it in a high-level language, and leverage the gained productivity to implement high-level performance-enhancing ideas first. If we ever run out of such ideas and converge to a stable architecture and feature set, we may decide to lower and specialize our code for optimal performance – but we would want to do this with the help of tools.

The reason for the use of C is in part historical. The first relational DBMS used C because it was the state of the art in programming languages for which good compilers were available. We never rethought our modus operandi sufficiently since. Today, the major players are invested in very large C codebases and armies of C programmers strongly bonded to their codebases. They have little choice but to continue.

Even in academic circles, building systems prototypes in C today is a matter of pride and not entirely rational.

THE BAR IS LOW. Isn't the correctness of the thesis of this article unlikely, given that the major commercial database systems are efforts of tens of thousands of expert years, and thus highly refined and efficient? Even in their advanced states, these code bases are in flux; feature requests keep coming in and algorithmic and architectural improvements keep happening, preventing the code bases from converging to that hypothetical state of perfect, stable maturity that can just not be beaten performance-wise. It has been argued, though, that the current DBMS codebases are rather elderly and tired anyway [SMA+07]. A new development from scratch, no matter whether in a high-level or low-level PL, would look very different.

More fundamentally, there are multiple reasons why even low-level DBMS source code must make ample use of indirection. Indirection (using procedures etc.) is necessary for basic code structuring, to keep the codebase from exploding into billions of lines of code. Manually inlining such indirections would make it impossible for humans to maintain and work with the codebase. Here, the human element of software development is in conflict with that last bit of performance a compiler might obtain. To illustrate the tradeoff between indirection and inlining, in a recent version of Postgres, there were seven B-tree implementations and at least 20 implementations of the page abstraction [Klo12], variously inlined with other behavior. This was certainly done for performance, which could be improved even further by more inlining, which becomes impractical to do manually.

A DBMS that implements the ANSI three-layer model needs indirection to realize the layering, for instance to support a dynamically changeable schema. Here, indirection is a *feature* of the system, but its performance-degrading effects could be eliminated by *staged compilation*, that is, *partial evaluation* to create code just-in-time that hardwires the current schema and eliminates schema-related indirections (cf. [RAC+14] for an example). This is easy to do using frameworks such as MetaOCAML [Met] or LMS [RO12], but impractical for C.

Thus the bar is low: There are many angles of attack by which a DBMS implemented in a high-level PL may outperform state-of-the-art DBMS.

[2] Compiler remedy: inlining.

[3] Compiler remedies: common subexpression elimination (CSE) and dead code elimination (DCE).

[4] Compiler remedy: depending on context, inlining, fusion, and deforestation.

DOMAIN-SPECIFIC LANGUAGES. Let us consider systems code written in a high-level imperative (such as Java) or impurely[5] functional programming language (such as Scala). There is an important special case that is worth discussing. Suppose we restrict ourselves to using dynamic datastructures only from an agreed-upon, limited library. A priori, there are no restrictions regarding which programming language features (and specifically control structures) may be used, but we agree not to define further dynamic data structures, avoid the use of imperative assignments other than of values of the provided data structures, and use collection operations such as foreach, map, fold, and filter to traverse these data structures rather than looping with counters or iterators. We can call such a convention a *shallowly embedded*[6] *domain-specific language (DSL)*. Natural choices of data structures to create a DSL around would be, say, (sparse) matrices and arrays for analytics, or relations for a PL/SQL style DSL. Thus, the unique flavor of the DSL is governed by the abstract data types it uses, and we do not insist on new syntax or exotic control structures.

AUTOMATICALLY ELIMINATING INDIRECTION. The relevance of the restriction to such a DSL will become apparent next. Let us look at which compiler optimizations we need to eliminate the indirections resulting from high-level programming discussed above. Some optimizations, such as duplicate expression elimination and dead code elimination, are generic and exist in some form in many modern compilers.

However, other key optimizations depend on the dynamic data structures. Most loops in programs are over the contents of collection data structures, so let us consider optimizing this scenario. Transforming a collection causes the creation and destruction of many objects; the consecutive execution of two such transformations causes the creation of an intermediate collection data structure. Such intermediate object creations and destructions can in principle be eliminated by loop fusion and deforestation.

Let us consider the case that we have an agreed upon interface for collections with a higher-order function map(f) that applies a function f to each element of the input collection c and returns the collection of results of f. Then we can fuse and deforest c.map(f).map(g) into c.map(g ∘ f). The optimized code performs a single traversal of the collection and produces the collection of results of the composed function g ∘ f applied to each member of c. The original code is less efficient because it in addition creates and destroys an intermediate collection c.map(f) along the way. Of course this transformation is only permitted if f and g have no side effects. In addition, the interface of collections must be known to the compiler. This example is admissible both for functional and imperative collections. However, particularly for the imperative collection implementations, the admissibility of this fusion optimization is too hard to determine automatically (by program analysis) if the compiler does not *know of collections*.

DOMAIN-SPECIFIC VS. GENERIC COMPILER OPTIMIZATIONS. For every possible transformation, a compiler implementor has to make a decision whether the expected benefit to code efficiency justifies the increase in the code complexity of the compiler. Optimizations that rarely apply will receive low priority.

The core of a collections interface using an operation map(f)[7] is known as a monad [BNTW95]. Monads play a particularly important role in pure functional programming languages such as Haskell, where they are also used to model a variety of impure behaviors such as state and exceptions, in addition to collections [Wad95]. The frequent occurrence of monads calls for compilers to be aware of them and to implement optimizations for them. Thus Haskell compilers implement fusion and deforestation optimizations for monads.

In compilers for impure and imperative languages, however, monads are unfortunately still considered exotic special cases, and such optimizations are generally not implemented.[8] This motivates DSLs constructed around abstract data types/interfaces on which algebraic laws hold that call for special optimizations. If the DSL is of sufficient interest, a compiler for this DSL can be built that supports these optimizations.

AUTOMATIC DATA STRUCTURE SPECIALIZATION. Once we decide to fix a DSL and build an optimizing compiler for it, it is natural to add further intelligence to it. One natural thing to do is to make the

[5] That is, which supports imperative programming when necessary.

[6] Shallow embedding means that the language extension is made through a library; no lifting of the language into an abstract syntax tree is required to be able to process the DSL using a mainstream programming language and compiler.

[7] This is imprecise in many ways, but because of space limitations, I just refer to [Wad95, BNTW95].

[8] Programmers are assumed to still prefer looping with iterators or counters over using such interfaces. This will change with time as a new generation of programmers grows up with frameworks such as LINQ [Mei11], map/reduce, and Spark, which impose monad-style interfaces.

DSL only support one intuitively high-level data structure and to make it easy to use by creating a high-level interface. Performance can then be gained by automatically specializing the data structure implementation in the compiler, decoupling the user's skill from the efficiency of the code. For example, a DSL may only support one high-level matrix data structure, and internally specialize it to a variety of low-level implementations and normal forms. A PL/SQL style DSL may offer a single relation abstraction, forcing the user to program on a high level without reference to index structures and such; the compiler may then introduce a variety of row, columnar, or other representations, as well as indexes. Of course, this separation of logical and physical data representation is a key idea in relational databases and very well studied. What compilers contribute is the potential elimination of all indirection overhead arising from this separation of concerns. In a main-memory database scenario, we deal with lightweight accesses to in-memory structures that can be inlined with the workload code, while an access to a secondary-storage (index) structure may turn into the execution of millions of CPU instructions.

EVIDENCE: THE CASE OF TPC-C. Above, we have covered two paths to performance (basic compiler optimizations and data structure specialization/index introduction), and discussed how a DSL compiler can implement both. But are these two groups of code transformations and optimizations really sufficient to do the job? In [SMA+07], the H-Store team went through an exercise that suggests the answer is yes. The paper describes an experiment in which expert C programmers implemented the five TPC-C transaction programs in low-level C, using all the tricks of their repertoire to obtain good performance. This code is used as a standard of efficiency in the authors' argument for a new generation of OLTP systems, and outperforms classical OLTP systems (which however have additional functionality[9]) by two orders of magnitude. An inspection of the resulting code [DCJK13] shows that the expertly written C code can be obtained by fusion, deforestation, and inlining, plus data structure specialization[10] from a naïve implementation in a high-level DSL of the type of PL/SQL. The paper [DCJK13] presents a DSL compiler that implements exactly these optimizations. The compiler generates low-level code from PL/SQL implementations of the TPC-C transaction programs. This code outperforms the hand-written C code of [SMA+07] but would have been equivalent if the programmers of [SMA+07] had applied their optimization ideas consistently and exhaustively.

THE KEY MANTRA OF DSL PERFORMANCE. This has been observed time and again in research on DSL-based systems [BSL+11, RSA+13, DCJK13, KRKC14]:

> *The central abstract data type shall be the focal point of all domain-specific optimizations.*

If we consider that, in the context of RDBMS, this just means *mind the relations*, it is really obvious. But the meaning of the mantra is this: Computations do not happen in a vacuum: They happen on data, and the bulk of this data is stored in values of the key abstract data type (relation, matrix, etc.). Such a value is not atomic; computations loop intensively over it, and in optimizing these loops (most of them of the map, fold, and filter varieties) lies the greatest potential for code optimization.

REVISITING THE SQL DSL. SQL is the most successful DSL yet, and the database community has amassed a great deal of expertise in making SQL-based DBMS perform. Leaving aspects of concurrency and parallelization aside, SQL databases draw their performance from (1) keeping the search space for optimizations under control and (2) leveraging domain-specific knowledge about an abstract datatype of relations (which ultimately governs representation, indexing, and out-of-core algorithms). We can relate (1) to the *functional* and *domain-specific* (and thus concise) nature of relational algebra.

Since SQL supports a universal collection data type (relations) and language constructs for looping over (FROM/joins) and aggregating the data, SQL systems can be used to embed many other DSLs by moderate extension. Consequently, we have studied SQL extended by UDFs, objects, windows, cubes, counterfactuals, PL control structures, probabilities, simulations, new aggregates for analytics, machine learning primitives, and many other features. By our mantra, SQL engines make a nontrivial contribution to the optimized execution of these DSL embeddings. However, in general, embedding a DSL into an SQL system in this way causes us

[9] Both the C code of [SMA+07] and the code generated by the compiler of [DCJK13] run in main memory and there is no concurrency control, latching, or logging.

[10] This comes in two forms, specialization of relations to arrays for tables that cannot see insertions or deletions during the benchmark, and the introduction of (B-tree) indexes to efficiently support certain selections in queries.

to miss optimization opportunities unless we make use of an extensible optimization framework that allows us to instruct the system how to exploit additional algebraic properties of the DSL. This could be an extensible query optimizer (cf. [HFLP89]) or an extensible DSL compiler framework such as EPFL's LMS [RO12, RSA+13], Stanford/EPFL's Delite [BSL+11] and Oracle's Graal/Truffle [WW12]. These projects provide infrastructure to easily and quickly build DSL compilers and efficient DSL-based systems.

DATABASE (QUERY) COMPILERS IN THE LITERATURE. The compilation of database queries has been studied since the dawn of the era of relational DBMS. IBM's System R initially used compilation techniques for query execution [CAB+81] before, even prior to the first commercial release of the system, converging to the now mainstream interpretative approach to query execution. Recent industrial data stream processing systems such as Streambase and IBM's Spade also make use of compilation. Microsoft's Hekaton [DFI+13] compiles transaction programs.[11] Furthermore, there have been numerous academic research efforts on compiling queries; recent ones include [RPML06], DBToaster [AK09, AKKN12], [KVC10], and Hyper [Neu11].

It is fair to say, though, that compiling queries is not mainstream today. The reason for this is that we have typically made our lives very hard for ourselves when creating compilers. System R abandoned compilation because it was done by simple *template expansion*, causing a productivity bottleneck [Moh12]. Rather than implementing an algorithm A, one had to implement an algorithm B that, when called, would generate the code of algorithm A – ultimately by string manipulation. This substantially slowed down development in a time when little was established about building RDBMS and much had to be explored. Code generation by template expansion of query operators is still practiced although decades behind the state of the art in compilers.[12]

We do not put enough effort into designing our compilers well. Often, building a compiler is a subservient goal to building a DBMS, and the compiler's design ends up being a hack. Focusing on just one DSL that has rather limited structure (queries) makes it hard to get our compilers right. Often, our compilers grow organically, adding support for new language constructs as the need arises or we discover a previous design to be flawed. With time this leads to unmanageable code that is ultimately abandoned.

This calls for a separation of concerns. A compiler should be developed for a well-defined general-purpose language, ideally by compiler experts; it can later be customized for a DSL.[13] In this article, I am arguing for compiling entire systems, not just queries. Thus the compiler must support a sufficiently powerful language. By aiming to build extensible compilers for general-purpose programming languages, we may be spreading ourselves too thin; fortunately, such compiler frameworks are now readily available [RO12, BSL+11].

Two observations regarding modern compiler technology that are valuable for people interested in building database compilers concern *lowering* and modern *generative programming*.

LOWERING. It is considered good practice to build a compiler that transforms intermediate representations (IRs) of programs by lowering steps, which transform IRs to lower and lower levels of abstraction (cf. e.g. [RSA+13]). Ideally, the final IR is at such a low level of abstraction and conceptually so close to the desired output code that code generation reduces to simple stringification. Compared to template expansion where DSL operations are replaced by monolithic code templates, breaking up the compilation process into smaller steps creates additional optimization potential and allows the type system of the PL in which the compiler is developed to guide the development process.

MODERN GENERATIVE PROGRAMMING. There is a rich set of ideas for making compiler implementation easier, from quasiquotes [TS00] to tagless type-based generative metaprogramming systems [CKS09, RO12] and multi-staging [Tah03]. This work makes use of the concept of compilation as *partial evaluation* in various ways (cf. [RAC+14] for an illustration of partial evaluation applied to compiling away the data dictionary).

Lightweight Modular Staging (LMS) [RO12] is a framework for easily creating optimizing compilers for DSLs. It is written in Scala and offers a Scala compiler as a library. Using Scala's powerful software composition features, it is easy to add domain-specific optimizations to this compiler. Scala-based DSLs do not even require

[11] Compared to [DCJK13], Hekaton performs less aggressive inlining (also with a view towards code size) and puts less emphasis on removing indirection. It mostly preserves relational plan operator boundaries and wires operators together by gotos [FIL14].

[12] Citations elided out of courtesy: the reader is invited to inspect the recent papers cited in the previous paragraph.

[13] Take LLVM/clang, which was or is used in a number of query compilation projects [AK09, Neu11, WML14, VBN14]. The clang compiler was done by compiler experts, but it is not easy to extend by domain-specific optimizations. Typically, code given to clang is too low-level for most of the collection optimizations discussed earlier [KRKC14].

the creation of a parser, because of Scala's ability to lift Scala code into an abstract syntax tree using language virtualization. LMS offers default code generators for Scala and C, and the Delite project [BSL+11] offers a parallelizing collections library and code generators for map/reduce and CUDA. In addition, the Delite team have created a growing number of DSLs (for querying, graph analytics, machine learning, and mesh computing) together with LMS-based compilers.

Staging in LMS refers to support of generative metaprogramming where representation types are used to schedule partial evaluation at different stages in the code's lifecycle. This allows to easily create programming systems that offer arbitrary combinations of static compilation, interpretation, and just-in-time compilation [RSA+13]. The usefulness of generative metaprogramming with LMS in the context of data management is illustrated in [RAC+14] using an example very similar to eliminating indirections due to a data dictionary in a just-in-time compiler. Using LMS, optimizing DSL compilers can typically be written in 3000 lines of code or less [BSL+11, AJRO12, DCJK13, KRKC14].

COMPILING SYSTEMS WITHOUT REGRET. There is increasing evidence that compilers can match and outperform human expert programmers at creating efficient low-level code for transaction programs [DCJK13] and analytical queries [KRKC14].[14] Admittedly, this does not yet "prove" the main thesis of this paper for complete database systems that deal with multi-core, parallelization, concurrency, and failures. Here, however, I point at systems such as Singularity and Spark that address such aspects and are implemented in high-level languages, with convincing performance. Ultimately, implementations of the aspects of systems that I have mainly brushed aside in this article spend much of their time in system calls. They depend less on low-level code efficiency and more on the choice of protocols and algorithms. Using a high-level PL, even unoptimized, is not much of a disadvantage [LH10].

An interesting goal for the future is to create a (database) *systems programming DSL*, adapting the ideas of work like [BGS+72] to the current millenium's changed perceptions of what constitutes high-level programming.

COMPILER-SYSTEM CODESIGN. There are now DSL compiler frameworks such as LMS that are easy to extend by (domain-)specific optimizations. Should we come to face a scenario in which our compiler produces locally bad code for our DBMS, we can always specialize our compiler by plugging in an ad-hoc transformation that deals with that particular issue at this place in the code. A general solution of the issue may not be easy to come by, but such a hack is. In terms of productivity, we profit from overall development of the DBMS in a high-level language and the fact that such hacks will be rare and will be needed in at most a few particularly performance-critical places in the codebase. We can eulogize such a strategy as *compiler-system codesign*. Isn't this a remedy to *any* unexpected obstacle to the truth of this article's thesis that we may yet encounter?

ONE FACTORY FITS ALL. Some researchers have criticized the commonplace "one size fits all" attitude in databases [Sto08] and have argued for the specialization of DBMS to render them well-matched to their workloads. Elsewhere, the argument is taken to call for the creation of one specialized system from scratch for each relevant use case. We should reconsider this and let compilers, rather than humans, do the leg work of specialization. Given a suitable library of DBMS components (query operators, concurrency control algorithms, etc.), there can be a single compiler-based factory for generating all these specialized database systems.

References

[AJRO12] S. Ackermann, V. Jovanovic, T. Rompf, and M. Odersky. Jet: An embedded DSL for high performance big data processing. In *International Workshop on End-to-end Management of Big Data*, 2012.

[AK09] Yanif Ahmad and Christoph Koch. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *PVLDB*, 2(2):1566–1569, 2009.

[AKKN12] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. In *VLDB*, 2012.

[BGS+72] R. Daniel Bergeron, John D. Gannon, D. P. Shecter, Frank Wm. Tompa, and Andries van Dam. Systems programming languages. *Advances in Computers*, 12:175–284, 1972.

[14] In [KRKC14], we implement an analytical query engine in Scala and use our LMS-based optimizing compiler to compile *it together with a query plan* to remove abstraction costs and obtain competitive query performance. These two papers are first steps in an ongoing effort at EPFL to build both an OLTP and an OLAP system in Scala to provide practical proof of this article's thesis.

[BNTW95] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.

[BSL+11] Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *PACT*, pages 89–100, 2011.

[CAB+81] Donald D. Chamberlin, Morton M. Astrahan, Mike W. Blasgen, Jim Gray, W. Frank King III, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, Patricia G. Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of System R. *Commun. ACM*, 24(10):632–646, 1981.

[CKS09] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.

[DCJK13] Mohammad Dashti, Thierry Coppey, Vojin Jovanovic, and Christoph Koch. Compiling transaction programs. 2013. Submitted for publication.

[DFI+13] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *SIGMOD Conference*, pages 1243–1254, 2013.

[FIL14] Craig Freedman, Erik Ismert, and Per-Ake Larson. Compilation in the Microsoft SQL Server Hekaton engine. *IEEE Data Engineering Bulletin*, 37(1), March 2014.

[HFLP89] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible query processing in Starburst. In *SIGMOD Conference*, pages 377–388, 1989.

[Jav13] Java 8: Project Lambda, 2013. http://openjdk.java.net/projects/lambda/.

[Klo12] Yannis Klonatos. Personal communication, 2012.

[Koc13] Christoph Koch. Abstraction without regret in data management systems. In *CIDR*, 2013.

[KRKC14] Yannis Klonatos, Tiark Rompf, Christoph Koch, and Hassan Chafi. Legobase: Building efficient query engines in a high-level language, 2014. Manuscript.

[KVC10] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010.

[Lam93] Butler Lampson. Turing award lecture, 1993.

[Lea13] Graham Lea. Survey results: Are developers more productive in Scala?, 2013. http://www.grahamlea.com/2013/02/survey-results-are-developers-more-productive-in-scala/.

[LH10] James R. Larus and Galen C. Hunt. The Singularity system. *Commun. ACM*, 53(8):72–79, 2010.

[Mei11] Erik Meijer. The world according to LINQ. *Commun. ACM*, 54(10):45–51, 2011.

[Met] MetaOCaml. http://www.metaocaml.org.

[Moh12] C. Mohan. Personal communication, 2012.

[Neu11] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[Pyt14] Python Programming Language Website. Quotes about Python, 2014. http://www.python.org/about/quotes/.

[RAC+14] Tiark Rompf, Nada Amin, Thierry Coppey, Mohammad Dashti, Manohar Jonnalagedda, Yannis Klonatos, Martin Odersky, and Christoph Koch. Abstraction without regret for efficient data processing. In *Data-Centric Programming Workshop*, 2014.

[RO12] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, 2012.

[RPML06] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled query execution engine using JVM. In *ICDE*, 2006.

[RSA+13] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Martin Odersky, and Kunle Olukotun. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *POPL*, 2013.

[SMA+07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[Sto08] Michael Stonebraker. Technical perspective - one size fits all: an idea whose time has come and gone. *Commun. ACM*, 51(12), 2008.

[Tah03] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, 2003.

[TS00] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[VBN14] Stratis Viglas, Gavin Bierman, and Fabian Nagel. Processing declarative queries through generating imperative code in managed runtimes. *IEEE Data Engineering Bulletin*, 37(1), March 2014.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52, 1995.

[WML14] Skye Wanderman-Milne and Nong Li. Runtime code generation in Cloudera Impala. *IEEE Data Engineering Bulletin*, 37(1), March 2014.

[WW12] Christian Wimmer and Thomas Würthinger. Truffle: a self-optimizing runtime system. In *SPLASH*, 2012.

[ZCD+12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.